Context Insensitive Escape Analysis (extending classical analysis with MHP)

Rohit Bhagat[1†] and Akshat Meena[1†]

[1]Department of Computer Science and Engineering, IIT Madras, Chennai, Tamil Nadu, India.

Contributing authors: cs19b038@smail.iitm.ac.in; cs19b052@smail.iitm.ac.in;
[†]These authors contributed equally to this work.

Abstract

This report focuses on the limitations of escape analysis[1] and proposes a solution to improve its results using May Happen in Parallel(MHP) analysis[2]. The classical escape analysis is too conservative and follows an all-or-nothing approach. MHP analysis can be used to reduce number of false positives, and the outcomes of escape analysis will be enhanced. The report explores different approaches from prior research on the topic. Additionally, the report proposes a new approach to improve escape analysis results by incorporating MHP analysis. MHP analysis helps to identify potential parallelism in the code, allowing for more accurate tracking of parallel accesses to an object and more precise escape analysis. Overall, this report provides valuable insights into the limitations of escape analysis and proposes a novel solution to improve its results using MHP analysis.

Keywords: Classical Escape Analysis, May Happen in Parallel Analysis

1 Introduction

In recent years, the demand for high-performance software has been on the rise. Developers are under increasing pressure to deliver applications that can process large volumes of data quickly and efficiently. In response to this demand, many optimization techniques have been developed to improve the performance of software applications. One such technique is escape analysis, which identifies objects that can be allocated on the stack rather than on the heap, reducing the overhead associated with memory management and unnecessary synchronizations, that can be eliminated to improve performance. However, classical escape analysis techniques have limitations, and their results can be inaccurate, resulting in missed optimization opportunities and degraded performance.

This report focuses on addressing the limitations of escape analysis and proposes a novel approach to improve its accuracy using May Happen in Parallel (MHP) analysis. MHP analysis helps to identify potential parallelism in the code, allowing for more accurate tracking of parallel accesses to an object and more precise escape analysis. Thus, better optimisations.

Overall, the report aims to provide valuable insights into the limitations of escape analysis and proposes a novel solution to improve its results using MHP analysis. The proposed approach has the potential to enhance the performance of software applications and improve the efficiency of the software development process.

1.1 Cassical Escape Analysis

The classical escape analysis technique involves analyzing the code of a program to determine whether an object is created within a specific scope or can be accessed globally. If an object is determined to have a limited scope and is not accessed outside its scope, it can be allocated on the stack, resulting in a more efficient memory management technique than heap allocation.

However, classical escape analysis may be less accurate when applied to complex software applications. It may inaccurately classify objects as having a global scope even when they are not accessed by other parts of the program, leading to missed optimization opportunities, reduced performance, and increased memory usage.
Classical escape analysis is widely used in contemporary programming languages such as Java, C++, and Go. It is particularly useful in optimizing applications with significant object creation and manipulation, such as web servers, database applications, and scientific computing applications. It is also used in embedded systems, where memory management is critical due to limited resources.
In conclusion, classical escape analysis is an effective technique for optimizing memory management and improving software performance. However, it has limitations, especially with complex applications. With software applications becoming more sophisticated, new approaches to optimize memory management are required. Researchers have proposed various techniques, including incorporating May Happen in Parallel (MHP) analysis, which have the potential to enhance the accuracy of escape analysis results and further improve software application performance.

1.2 MHP Analysis

May Happen in Parallel (MHP) analysis is a technique used in computer programming to identify potential concurrency bugs in multithreaded programs. This method examines a program's code to find instances where two or more instructions might run concurrently.
MHP analysis involves identifying pairs of instructions that may execute concurrently, regardless of whether they actually do execute concurrently. This analysis is based on the observation that if two instructions can execute concurrently, there is a possibility that they may interfere with each other and cause a concurrency bug.
Potential concurrency issues in multithreaded programmes are found via MHP analysis. Because it might be difficult to find and fix these flaws, MHP analysis is a crucial tool for engineers. Applications that use a lot of parallelism, such web servers, scientific computing programmes, and data processing programmes, can benefit greatly from MHP analysis.
MHP analysis can be used to improve classical escape analysis by identifying situations in which an object may be accessed concurrently by multiple threads. In such situations, the object cannot be allocated on the stack, as it may be accessed after it goes out of scope. By incorporating MHP analysis into classical escape analysis, developers can improve the accuracy of escape analysis results and reduce the risk of concurrency bugs.

1.3 An overview of the report

The report is organized as follows. Section 2 addresses some limitations and their possible solution using MHP analysis information. Section 3 provides insights on the prior work done on this topic, it briefly explains approaches and concepts mentioned in these papers. Section 4 introduces the proposed approach to improve escape analysis results using MHP analysis. Section 5 presents a comparison between the output of classical escape analysis and the output obtained using the proposed approach. The comparison is illustrated through several examples to demonstrate the improvements achieved through the use of the proposed approach.

2 Limitations

As the classical escape analysis uses all-or-nothing approach meaning that an object either permanently escapes or permanenly not escapes. This approach is too conservative and thus gives false positives. Some of the circumstances that shows limitations of classical escape analysis are:

1. In some cases such as an object can be accessed by more than one thread but only one thread is present in the entire program execution hence no more than two threads will access it in parallel, so the object is not escaped.
2. If synchronization lock on an escaped object is taken in two different parts of program but they are not running in parallel then by classical escape analysis results we cannot eliminate the synchronize block. Giving us a false positive.
3. If two program block(having a shared variable) are running in parallel and an object from one thread escapes but there is no read/write on the object in other thread. The object escapes according to classical escape analysis but in truth we can treat it as not escaped and optimise the code accordingly.

4. If two program block(having a shared variable) are running in parallel and an object from one thread escapes but the shared variable is assigned to some other object subsequently. And there is no read/write on the previously escaped object before the shared variable is overwritten. Classical escape analysis says the object escapes but the object was reclaimed before it was accessed thus considered false positive.

Respective possible solutions using MHP analysis information are:

1. Using MHP information we can determine threads running in parallel and also determine shared variables between them which we can analyse to check if an object really escapes.
2. We can use MHP information to determine synchronization blocks runnning in parallel and which are not and eliminate synchronization blocks accordingly.
3. MHP analysis results can be used to determine all the nodes that can possibly be executed while the escaped object is in the scope. We can then check respective read/writes.
4. Same solution as for 3rd.

3 Prior Work

In the paper titled "Pointer and Escape Analysis for Multithreaded Programs"[3] by Alexandru Salcianu and Martin C. Rinard some approaches are proposed to apply MHP info in escape analysis. This paper introduces a novel approach to analyze multithreaded programs through a combined pointer and escape analysis. The proposed algorithm employs a new abstraction called parallel interaction graphs to examine the correlations among threads and accurately extract precise information about points-to relationships, escape analysis, and the order of actions pertaining to objects accessed by multiple threads.

3.1 Parallel Interaction Graph

A parallel interaction graph is a tuple $\langle\langle O, I, e\rangle, \tau, \alpha, \pi\rangle$:

- The thread set $\tau \subseteq N$ represents the set of unanalyzed thread objects started by the analyzed computation.
- The action set α records the set of actions executed by the analyzed computation. Actions are like synchronize blocks, read/write, invoking method.
- The action order π records ordering information between the actions of the current thread and threads that execute in parallel with the current thread.

3.2 Relevant Results

In contrast to earlier synchronization elimination algorithms, their approach goes beyond and investigates the interactions among parallel threads. Subsequently, it leverages the action set denoted as α and the action ordering relation represented as π to remove synchronizations objects with synchronizations from multiple threads.
The analysis process is as follows: After conducting the interthread analysis, each captured node n is examined. The analysis checks the action ordering relation π to identify all threads t that run in parallel with a synchronization on n. Subsequently, the action set α is assessed to determine if thread t also synchronizes on n. If none of the parallel threads t synchronize on n, the compiler has the capability to eliminate all synchronizations on the objects represented by n. Even in cases where multiple threads synchronize on these objects, the analysis has determined that the synchronizations occur at distinct points in time due to thread start events, rendering them redundant.

4 Our Approach

For our approach we are assuming that we already have the required MHP information.
We are dividing our approach in 3 parts:

1. **Confirm** - This steps confirms if the thread object may be accessed in any parallel thread or not before setting the object as escaped.
    - Let N denote all the nodes of the threads running in parallel to t_1.
    - Let $W \subseteq N$ denote nodes with read/write on the escaped object.
    - If $W = \phi$ then the object doesn't escapes otherwise it escapes.

2. **Reclaim** - In this step we try to find if there is some object that has escaped from a thread but can be reclaimed after sometime (set to not escaped) without affecting the validity of analysis.
   - Let O_e be an escaped object from thread t_1.
   - Let g be the shared variable such that $O_e \in \rho(g)$.
   - Let N denote all the nodes present in t_1 and the threads running in parallel with t_1.
   - Let $S \subseteq N$ represent the set of nodes denoting assign operation to g.
   - Let $W \subseteq N$ be the set of node with read/write action on the escaped object O_e.
   - Then, if $\exists n \in S : n \; HB \; w, \quad \forall w \in W$ (Happen Before relation) then we can reclaim the escaped object O_e.
3. **Pruning** - This step is performed after the escape analysis is done to further refine the results. Using this result we can eliminate synchronization blocks using MHP information.
   For every synchronization node(n), taking lock on variable pointing to set of objects *O(n)*:
   - Let the set of synchronization nodes running in parallel with n be $S(n)$
   - If $(\cup_{m \in S(n)} O(m)) \cap O(n) = \phi$ then we can remove the synchronization block.

The first two steps are done in parallel with the classical escape analysis assuming we already have the MHP information. Third step is done after classical escape analysis is done.

The Reclaim step can be further extended in the circumstances where the escape object is stored in a thread local variable l of some parallel thread. In that case we have to add all the nodes corresponding to the assign statements for l in the set S. But this extension can result in exponential increase in the analysis making it less feasible.

5 Examples

```
class A {
    static A a, b;
    int j;

    public static void main(...) {
        A t = new A(); // O1
        A u = new A(); // O2
        A v = new A(); // O3
        A w = new A(); // O4
        A x = new A(); // O5
        B y = new B(); // O6
        D z = new D(); // O7
        C.c = t;
        synchronized (t) { // O1
            // ... some code
        }
        D.d = u;
        z.start();
        D.d = v;
        synchronized (u) { // O2
            z.i = 1;
            D.d.notifyAll();
        }
        z.join();
        a = w;
        b = x;
        y.start();
        synchronized (w) { // O4
            // ... some code
        }
        y.join();
        synchronized (x) { // O5
            // ... some code
        }
    }
}

class B extends Thread {
    public void run() {
        synchronized (A.a) { // O3
            // ... some code
        }
        synchronized (A.b) { // O4
            // ... some code
        }
    }
}

class C extends Thread {
    static A c;

    public void run() {
        Boolean d = false;
        if (d) {
            System.out.println(10);
        }
    }
}

class D extends Thread {
    static A d;
    int i = 0;

64    public void run() {
65      while (i == 0) {
66        d.wait();
67      }
68      d.j = i;
69    }
70  }
```

If we apply classical escape analysis in this then:

- Escaped Objects = $\{O1, O2, O3, O4, O5, O6, O7\}$
- Synchronization blocks we can eliminate = None

Using proposed approach:

- O1 at line 13 doesnt escape using 1st step(Confirm). There is no parallel node running that share the shared variable *"C.c"*
- O2 escapes at line 17 but is reclaimed at line 19. Using the "Reclaim" step as there is HB relation between the write on line 68 and assign statement on line 19.
- O3 escapes at line 19.
- O4 escapes at line 25.
- O5 escapes at line 26.
- In comparision to the classical escape analysis, the objects O2 and O1 doesnt escape thus we can remove synchronization blocks at line 14 and line 20.
- Using "Pruning" after analysis, we can see that synchronization block at line 32 and line 43 can be removed as they are not in parallel with each other.

As we can see we have improved our result using MHP analysis information

6 Conclusions

The classic escape analysis is too conservative and thus give false positives in many cases as it follows all-or-nothing approach. Using MHP information we were able to make the escape analysis more precise. We can employ the MHP information during and after the escape analysis to improve the results as seen from the proposed approach. We can further increase the results by employing further complex analysis using MHP information but such approach can be less feasible if we consider time complexity for the optimisation.

References

[1] Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V., Midkiff, S.: Escape analysis for java. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA **34** (2000) https://doi.org/10.1145/320385.320386

[2] Naumovich, G.: An efficient algorithm for computing mhp information for concurrent java programs (1999). Computer Science Department Faculty Publication Series. 104

[3] Alexandru Salcianu, Martin C. Rinard: Pointer and escape analysis for multithreaded programs (2001). Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2001)